SANDIA REPORT

SAND2008-7593
Unlimited Release
Printed November 2008

Trilinos CMake Evaluation

Roscoe A. Bartlett, Daniel M. Dunlavy, Esteban J. Guillen, Tim Shead

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2008-7593
Unlimited Release
Printed November 2008

Trilinos CMake Evaluation

Roscoe A. Bartlett, Daniel M. Dunlavy, Esteban J. Guillen, Tim Shead

Abstract

Autotools is terrible. We need something better. How about CMake? Here we document our evaluation of CMake as a build system and testing infrastructure to replace the current autotools-based system for Trilinos.

Acknowledgment

The authors would like to thank Kitware and the open-source community for creating such a great set of tools.

Contents

Appendix

1 Introduction

Here, we summarize a detailed evaluation of the CMake set of build and testing tools as a possible replacement for the current autotools-based and home-grown Perl-based test harness. This evaluation includes the development of a comprehensive prototype build system infrastructure for Trilinos using CMake that is likely very representative of what a final system would look like.

The CMake set of configuration, build, test, and reporting tools are developed and maintained by Kitware, Inc.[1] The four (4) tools available in the current CMake system are as follows:

- *CMake*: portable build manager that includes a complete scripting language for configuring and building software libraries and executables.
- *CTest*: executable to handle running tests along with a scripting language to control test flows.
- *CPack*: cross-platform software packaging tool, with installer support for all systems currently supported by CMake.
- *CDash*: build and test reporting dashboard built on PHP, CSS, XSL, MySQL, and Apache HTTPD.

Note that although these tools have been developed to work in tandem as a complete build and test environment, the components can be used independently as well. For example, CMake can be used to configure and build a library, and testing can be handled using another set of tools. Furthermore, CTest and CDash can be used to handle testing and reporting of a software library without using CMake as the build manager. This said, it is recommended to use the full CMake/CTest/CPack/CDash system as there is more work involved to use only parts of the system.

[1] www.kitware.com

2 Build Capabilities and Features

Here we describe a set of features relating to building and how they are (or are not) supported with the current autotools build system and the new CMake prototype system. These features are subdivided into critical and non-critical categories.

2.1 Critical build features currently handled by the existing autotools system

1. Dependency tracking of header files to rebuild object files:

   (a) Why this is important: To enable fast, safe, and correct rebuilds.

   (b) Autotools: Only supported on Linux and very few other platforms. Not supported on the Sun, IBM, and SGI.

   (c) CMake: Built into CMake for C and C++ so it will work on every platform, period!

2. Dependency tracking of object files to libraries:

   (a) Why this is important: To enable fast, safe, and correct rebuilds.

   (b) Autotools: Fully supported on all platforms.

   (c) CMake: Fully supported on all platforms.

3. Scalable build system where Trilinos package libraries and header file directory locations are only defined once:

   (a) Why this is important: This is needed to allow for the scalable growth and maintenance of Trilinos.

   (b) Autotools: The current export makefile system (with makefile fragments Makefile.package.export) defines a single file where these things are defined. Every test and example in Trilinos (with a few shameful exceptions) uses the export makefiles to get all of these. There is no duplication.

   (c) CMake: The prototype CMake-based build system for Trilinos handles all intra-package dependencies automatically with only simple intra-package dependency lists in Dependencies.cmake files. There is no duplication at all and this system is much cleaner than the current autotools system.

4. Build system must be portable to all required platforms:

   (a) Why this is important: Trilinos must support a wide set of different platforms.

   (b) Autotools: Autotools generates very portable makefiles. However, avoiding “command-line too long” errors on many platforms requires that one use GNU Make.

   (c) CMake: One needs to have cmake built on the target platform but all that is needed to build and install cmake is a fairly recent C++ compiler. A simple ’configure’, ’make’, and ’make install’ is all that is needed.

5. Cross compiling:

(a) Why this is important: Cross compiling is required to build for production parallel machines (e.g. Red Storm).

(b) Autotools: Yes, because everything is manual.

(c) CMake: Cross compiling is a new feature in CMake and it is unclear how well tested this is in the Sandia environment. However, organization 1420 has had some success with using this feature with Paraview to build for reddish/redstorm.

2.2 Critical build features not currently handled by the existing autotools system

1. Library to executable dependency tracking:

   (a) Why this is important: We need this for fast rebuild/retest cycles before repository commits and faster feedback from continuous integration builds. Currently, it takes up to 50 minutes or longer on a very fast Linux machine to relink a large number of the tests and examples in Trilinos.

   (b) Autotools: No support at all.

   (c) CMake: This is built into CMake. This is one of the major motivations for going to CMake.

2. MS Windows support:

   (a) Why this is important: Windows represents a big growth area for Trilinos; the Titan project is one example.

   (b) Autotools: No direct support but cygwin can be used with Windows Intel compilers to install headers/libraries. This is not easy to set up and not very portable.

   (c) CMake: Direct support for MS Windows and a variety of other systems. This includes creating project files for MS Visual Studio and binary installers using NSIS[2]. There is lots of experience with this at Sandia for Visual Studio and NMake for native Windows binaries. This is one of the major selling points for CMake over the autotools system.

3. Support for multiple languages including Fortran 77, Fortran 90, and Fortran 2003:

   (a) Why this is important: ForTrilinos requires a very current Fortran 90 compiler with some Fortran 2003 features. Included in this is name mangling for Fortran 77.

   (b) Autotools: This is currently handled by setting 'F90' to point to the Fortran 90+ compiler when calling configure. However, Fortran 2003 with gfortran creates auxiliary files that need to be symbolically linked in a manual way and it is not portable. Basically, autotools really does *not* support Fortran 2003. What we have right now is a hack that works with gfortran. Name mangling with Fortran 77 is well supported and largely automatic on most platforms.

   (c) CMake: It is unclear how well CMake supports Fortran 90+ but Chapter 11 in [1] deals with how to add new compilers so it seems this could be supported more cleanly than with autotools. One idea is to copy and extend the Fortran 90 macro package for Fortran 2003. This may be hard to make portable but hopefully we would get some help from the CMake community or Kitware. Right now Fortran 77 name mangling is determined by platform but it will be easy to write a Module/Macro that will figure this out automatically with the `try_compile(...)` command.

[2] A scriptable win32 installer/uninstaller system available at `http://nsis.sourceforge.net/`

2.3 Less-critical but desirable build features

1. Shared libraries:

   (a) Why this is important: Some customers demand this. Also, shared libraries can allow for much faster rebuild cycles where executables do not even need to be relinked after a dependent library is rebuilt. This can substantially speed up pre-checkin testing and shorten the feedback time from a continuous integration server.

   (b) Autotools: Past attempts at using libtool have failed. The Python-based system for creating shared libraries on Linux is not well characterized.

   (c) CMake: Shared library support with CMake is advertised to be very strong (see Section 4.8 in [1]). Experience that organization 1420 has had with shared libraries with CMake in our computing environment at Sandia includes Linux, Mac OS, and Windows (but Windows requires some extra work in the CMakeLists.txt files). If shared libraries don't work on a particular platform, you can just switch back to static libraries.

   Caveat: Currently, at least for the makefile generator, changes in `*.cpp` files that result in the recreation of a shared library also result in unnecessary relinking of all dependent test and example executables. This is unfortunate but understandable given that makefiles are being used and this type of special shared library logic is not supported by 'make'. *Work around*: We can add a 'libs' target that will result in only libraries being rebuilt and then a developer can just use 'make libs' followed by 'ctest' and avoid relinking. However, if other code related to a test/example also needs to be rebuilt, then that will result in inconsistent executables and therefore inaccurate test results. But, it may be possible to devise a general system of manipulating time stamps when shared libraries are used so that we can avoid relinking executables unnecessarily.

2. Creation of export makefiles with compilers, compiler options, include directories, libraries etc. for use by external client makefiles:

   (a) Why this is important: Building compatible client code that will successfully link against Trilinos can be difficult to do in a portable when done manually. This feature is really only used by second-tier customers (e.g. graduate students and other off projects) as our major customers handle this differently.

   (b) Autotools: This is done by installing Makefile.package.export and Makefile.package.export.macros in the $prefix/install directory.

   (c) CMake: This will be easy to support, and a prototype system is already in place for the LSALIB library currently used in the Titan project.

3. Clean and understandable mechanism for extending functionality:

   (a) Why this is important: We will need to create specialized features form time to time, and we need a supported way to do this that is clean and understandable.

   (b) Autotools: Autotools uses the so-called M4 language which has to be the most confusing and hacked language ever designed in semi-modern times. It is clear that M4 was hacked together in a pinch as some wrapper around shell scripting.

(c) CMake: CMake has its own scripting language that is fairly general and fairly compact for basic usage. It is a "Turing complete" (i.e. you can write complete programs with it) which includes the ability to define lists and loops over those lists and create arbitrary new functions (call macros). Unlike makefiles, commands in CMake script files are executed from beginning to end with clear programming language semantics. While there is no CMake "debugger", you can effectively debug CMake scripts by putting in `MESSAGE(...)` calls which are the equivalent to print statements.

Disclaimers: The CMake scripting language is a little strange. For one, the variable scoping rules involving cache, internal cache, and non-cache variables are a little confusing and make it difficult to implement complex logic. I would not try to write a complex stand-alone program or complex logic in CMake (but you could as all of the tools for doing so are there). However, CMake is still many times better than autotools for which we are comparing.

4. Simple development environment:

   (a) Why this is important: Leaning how to develop in a Trilinos package needs to be fairly easy and setting up a development environment needs to be easy as well.

   (b) Autotools: In order to develop on Trilinos, you need to go out and find specific versions of autoconf and automake and put them on whatever system where you will need to change the build system for any reason (e.g. add/remove files, add new tests, etc.). This is a big hassle when you work on a variety of platforms, especially ones where you do not have root administrator privileges (e.g. the SCICO LAN). Also, whenever you changed any Makefile.am file or configure.ac file, you need to remember to run ./bootstrap in any package that you change and at the top level. It is easy to forget and a hassle to run these scripts. And if you don't remember to bootstrap, the build system will not update anything. This is a real problem, especially for new developers.

   (c) CMake: With CMake there is no intermediate "bootstrap" step. After the CMake files are modified, you can just type 'make' and the system will reconfigure itself if it needs to do so. Also, there are no intermediate files that need to be added to the VS repository. As a developer, this means only having to get the CMake source and install it (version 2.6 is currently required but 2.7 is needed for correct CDash reporting). That is better than having to get two different programs with different versions for automake and autoconf.

5. Simple installation for users:

   (a) Why this is important: While installing Trilinos for users for the general case will never be "easy" because of the complexity of Trilinos, we don't want to make it harder than it needs to be.

   (b) Autotools: With autotools, a "configure" shell script is provided and in theory, a user just needs to run configure with the correct input arguments and then they can do 'make install'. Testing a serial version of Trilinos is also as easy as 'make runtests-serial'. Running the full serial test suite requires 'make runtests-serial TRILINOS_TEST_CATEGORY=FRAMEWORK'. However, running the MPI tests is not so easy, and we imagine most users will never do this because of having to specify several other tricky input 'make' variables like TRILINOS_MPI_GO (which requires double quotes). Also, in order to really install and test all of Trilinos on some systems, you have to use –with-gnumake or you get "command-line too long" errors or errors that are much more cryptic. What this means is that you essentially need

to have and use GNU Make in order to make Trilinos portable. On some systems, user may have to install GNU Make themselves but usually it is already installed and they just need to use it instead of the default 'make' command.

(c) CMake: Users must have a current enough version (currently 2.6) of CMake installed on their system to configure and generate makefiles and then build and install. However, for some specialized distributions, it will be possible to distribute Trilinos as RPM files, MS Visual Studio project files, and the like where users would not need CMake. However, this will result in less flexibility in what packages and options are enabled and disabled. Note, however, that CMake appears to be as portable and easy to build from source as GNU Make so this is not a huge extra requirement over the autotools system. Also note that on MS Windows, there are binary self-extracting installers for a very nice CMake GUI program that users can just download, double-click to install, and then run. On MS Windows, if you have the free MS Visual C++ express edition, then configuring project files by downloading and installing the Windows version of CMake is really not any harder than downloading a pre-created project file.

6. Circular dependencies allowed between tests/examples in different packages:

(a) Why this is important: It is not clear where to put some examples/tests which require multiple packages and there are several cases of tests/examples that refer to libraries that build later in the build tree.

(b) Autotools: The default target for 'make' just builds libraries and the 'tests', 'examples', and 'everything' targets build all libraries before any tests or examples are linked. The support for this with the current autotools-based system is less than straightforward. Also, support for these types of circular dependencies has proved to make the current autotools-based system quite fragile is not scaling well.

(c) CMake: The current CMake build system prototype does not support circular dependencies between tests/examples and later package libraries. However, it is not clear that this feature is worth preserving. Instead, we can put such troublesome tests/examples into packages like TrilinosCouplings (and NonlinearStrategies) or others that already have all of the needed package dependencies. Also, packages have used this feature as a crutch to avoid having to build appropriate mock objects in order to do proper unit testing.

3 Testing and Reporting Capabilities and Features

In this section we list some features that are needed for a testing system for Trilinos and compare current support for them in the autotools-based and CMake-based Trilinos systems. These features are separated into critical and less critical sets.

3.1 Critical testing features currently handled by the existing Perl-based test harness

1. Separate archiving and reporting of test results for each package:

   (a) Why this is important: It is important to create targeted package test result web pages and emails so that individual package developers can focus on their own packages and not be distracted by errors from other related packages. This is something that SIERRA does *not* have worked out yet and it is causing the problems.

   (b) Perl test harness: Handled very cleanly but there are still some improvements to be made. Examples of needed improvements: a) listing of all platforms run, b) email notifications of specific platform builds.

   (c) CMake/CTest/CDash: This can be done at the CTest scripting level and with some PHP database programing. This might take a significant amount of work to accomplish.

2. Platform-specific tests:

   (a) Why this is important: One must be able to target and specialize tests for different platforms in order to deal with portability problems and other issues.

   (b) Perl test harness: Handed with 'HOST' and 'X-HOST' options in a very clean way with 'uname -n' but it is not very portable (for example, does not work on the MAC).

   (c) CMake: This is trivially handled with a HOST and XHOST argument with the TRILINOS_ADD_TEST(...) function. Actually, this is more portable since it relies on the built-in SITE_NAME(...) command in CMake and not the non-portable 'uname -n' command.

3. Disabling of packages that fail the build and then rebuilding

   (a) Why this is important: In order to maximize the amount of (Experimental) code that can be built and tested, it is desirable to be able to disable packages that fail to compile and then disable optional support in the other packages.

   (b) Perl test harness: This is supported but it requires a lot of manual work to maintain the package dependencies and the completion banners in every package. This system is also fairly fragile and breaks a lot giving false results and resulting in code not being tested for days.

   (c) CMake/CTest: There is no built-in CMake support for this but intra-package dependence tracking has been implemented in the prototype CMake build system for Trilinos. An advanced CTest script (see Section 10.9 in [1]) could be written to drive the entire process and the dashboard display could be modified to support this. Because all intra-package dependency management would be handled automatically, this system would be much more robust than the current autotools Perl-based test harness.

4. Selection of subsets of tests using keywords:

(a) Why this is important: 1) routine developer testing during the development process requires the ability to run subsets of tests easily, 2) tests for individual packages need to be selected individually, 3) different sets of test categories like “Performance” and “Scalability” tests must selected as needed.

(b) Perl test harness: There is only superficial support where only one “keyword” can be selected for inclusion. However multiple sets of keywords can not be selected and keywords can not be excluded.

(c) CMake/CTest: While CTest does not directly support keywords, they are emulated with the TRILINOS_ADD_TEST(...) function by simply appending the keywords to the name of the test. Then, ’ctest’ supports the options -R, -E, and -U for including and excluding tests in a fairly flexible way.

3.2 Critical testing features not currently handled by the Perl-based test harness

1. Code coverage:

   (a) Why this is important: This is one of the most basic metrics of code quality and of the completeness of tests.

   (b) Perl test harness: Has been supported in the past but not currently.

   (c) CMake/CTest: Built-in support and run on many platforms by lots of groups.

2. Memory usage testing (i.e. Valgrind and/or purify):

   (a) Why this is important: Memory usage errors in C/C++ continue to degrade the quality of our code and they tend to sit dormant for long periods of time and don’t cause major problems until we really need our software to work. This includes memory leaks, accessing deleted memory, accessing memory wit.h invalid addresses, out-of-bounds errors, etc.

   (b) Perl test harness: Has been supported in the past but not currently

   (c) CMake/CTest/CDash: Built-in support and is run on many platforms by lots of other groups. This can be run in local build with:

   ```
   ctest -T memcheck
   ```

   It can also run memory checking as part of the nightly testing process with:

   ```
   ctest -D NightlyMemoryCheck
   ```

   Individual memory problems are cleanly reported on the CDash dashboard.

3. Automatic timeout of tests:

   (a) Why this is important: Hanging tests can freeze up the test harness so that no results are reported at all and it requires a lot of manual work to monitor this and to manually kill hanging tests. Every major test harness used by the Trilinos application customers have support for this feature (e.g. Alegra, Charon, SIERRA, Xyce).

(b) Perl test harness: Not currently supported.

(c) CMake/CTest: Currently supported with the CMake cache variable `DART_TESTING_TIMEOUT:STRING=<seconds>`. A timeout can also be set on a test-by-test basis with the `TIMEOUT` test property set by the built-in `set_tests_properties(...)` CMake command. This feature has been tested and verified to work well even with parallel tests (with Open MPI). This will result in a huge improvement in the robustness of the testing infrastructure for Trilinos.

3.3 Less-critical but desirable testing and reporting features

1. Performance testing:

   (a) Why this is important: Performance tests are typically serial tests that do relative or absolute run-time comparisons for optimized code. We need a mechanism for defining and selecting to run performance tests on various platforms for optimized builds.

   (b) Perl test harness: No direct support but could be handled with a special test category (i.e. keyword) and run for optimized builds?

   (c) CMake/CTest: Can be very easily supported with a CATEGORIES option with the TRILINOS_ADD_TEST(...) function.

2. Parallel running of (serial) tests:

   (a) Why this is important: Running tests is parallel can significantly speed up pre-checking testing and shorten the feedback time from a continuous integration server.

   (b) Perl test harness: No support.

   (c) CMake/CTest: This is being developed in the current CMake CVS development version (but does not seem to work yet).

3. PBS-type batch running of MPI tests:

   (a) Why this is important: Scalability testing requires some more substantial parallel clusters and this requires using a batch system like PBS. The test harness needs to support submitting batch jobs of MPI runs and wait for the results to come back in an efficient way.

   (b) Perl test harness: No support yet but some experimentation has been done.

   (c) CMake/CTest: Not directly supported but given the flexibility of the CMake scripting language, if it is possible to support, then this can be supported in the TRILINOS_ADD_TEST(...) function in a way that is largely transparent to the Trilinos developer. For example, the TRILINOS_ADD_TEST(...) function could add an initial PBS submit script with a call to ADD_TEST(...) and then a back-end could store a list of followup commands that would all get added after all initial tests are defined with additional with ADD_TEST(...) calls at the end to poll for completion of the various PBS jobs. With CMake, all of this can be handled automatically in a consistent way.

4. Archiving all test outputs for sufficient periods of time:

(a) Why this is important: Complete test results are needed to be able to diagnose failing tests. Otherwise, you must manually go to the platform, build the executable(s) and run the tests manually. Older test result data can be cleaned out as needed to make space.

(b) Perl test harness: Currently, only stdout is captured and saved and then only a limit of so many bytes for each file.

(c) CMake/CTest/CDash: CTest/CDash supports grabbing "measurements" including files and putting them in the CDash database. You can also post files to the dashboard database using the -A option with ctest.

- TODO: Look into ctest -A option for posting multiple files to dashboard. The VTK dashboard already has examples of this

We could also augment the system to store larger files in a separate directory structure outside of the database and then just put in HTML links from the posted CDash files. The new SIERRA Dart Dashboard system uses a system like this. We would then implement a separate job to clean out older results based on various criteria. This has been done, for example, for the SIERRA + Trilinos Integration testing scripts (i.e. the STANA scripts).

5. Allowing the specification of any arbitrary number of programs and/or criteria to determine the success or failure of a test:

(a) Why this is important: Complex tests require that you be able to define "success" in a variety of different ways. Examples: a) Grepping an output file looking for a specific string, b) checking for a non-zero return value and grepping for a specific string in stdout, c) running multiple test executables and then comparing files to define an overall test that gets reported (This could also be used for scalability testing for PBS-type queuing systems).

(b) Perl test harness: The current Trilinos test harness only allows you to run one script as the test or a single grep of the console output. There is also support for running the compareOutput program but that is not enough.

(c) CMake/CTest: No direct support but you could do this behind the scenes of the TRILINOS_ADD_TEST(...) function in a way that was 100% transparent. For example, the TRILINOS_ADD_TEST(...) function or a new similar function could handle multiple COMMAND and ARGS fields and on the back end could write a portable python script that would then be directly set by the built-in ADD_TEST(...) CMake command.

4 Desired enhancements to CMake/CTest/CDash

Here we list some identified areas of missing support in CMake/CTest/CDash that we either strongly need or would be of great benefit for Trilinos. These are features that could not easily be built over top of existing support or would greatly benefit from direct CMake support.

1. *Strongly desired*: CMake/CTest: Support for keywords for tests. This could be added with a `KEYWORDS` property for the `set_tests_properties(...)` command. *Work around*: Simply append all of the keywords to the name of the test.

2. *Desired*: Direct support for running (multiple) programs to post-process the output from a test (both the console and any output files). *Workaround*: This can be emulated within the TRILINOS_ADD_TEST(...) function by writing script files that combine everything but it will be hard to make this portable and might make it confusing to trace what is happening.

3. CTest: Default outputting issues:

   - *Desired*: Show what the test criteria is and why a test passed or failed in the test output in the output file Testing/Temporary/LastTest.log. Currently, it does not show if the test passed or failed, just the command used to invoke the test and the test output. *Workaround*: Run 'ctest -VV' and skip the shorter summary output. However, the absences of the summary output is greatly missed.
   - *Desired*: Print the CPU time for each test (not just the start and end times in the log file). The test run time helps to determine what tests are taking too long and need to be revised. This feature will be easy to add to the CTest C++ source code and we can do this ourselves.
   - *Desired*: Automatically widen the main summary output to show the full test names. Currently, only the first 30 characters of the name are shown. The option -W was added to the CVS version of CTest to allow the width to be manually set but this is a hassle and does not interact well with MS Visual C++ projects. This will be fairly easy to add to the CTest C++ source code.

4. *Desired:* CTest/CDash: Submit all test data (no matter the size) and then prune test results over time. This includes files that get output as well. This is being done for the SIERRA + Trilinos Integration test repository for instance. *Workaround*: We can emulate this with our own handling scripting code but this will not be available to other CMake projects.

5. *Desired*: CMake: Generate error messages for missing source files that have line numbers in the corresponding CMakeLists.txt file. Currently, it just lists the entire CMakeLists.txt file and nothing else. *Workaround*: Just add files slowly and re-run CMake each time to debug the problem.

6. *Desired:* CMake: Strong checking for variables that are not defined. Just letting undefined variables be empty is a bad practice (used by Make and bad Fortran). This practice is well known to result in higher rates of software defects. *Workaround*: Use a user-defined `ASSERT_DEFINED(...)` macro (which is being done right now and is somewhat effective).

7. *Desired:* CMake: Strong checking for user input misspelling CMake cache variables: Currently, if a user misspells the name of a defined user cache variable, the default for that variable will be used instead and the misspelled user variable will be ignored. This is very bad behavior that is carried over from the autotools world and should not be repeated with the CMake system. It would be very

useful if the cmake executable could take a new option (e.g. –validate-cache-variables) that would force the validation of all user-set cache variables to make sure that they had a matching internally defined cache variable. *Workaround*: We could create a new VALIDATED_OPTION(...) command that would store a list of all defined options and then we could devise a system that would validate that all cache variables set were expected. However, this would only work for Trilinos-defined variables and not other cache variables defined by built-in CMake commands like SITE_NAME(...) and FIND_FILE(...). User input checking is a serious software verification issue that needs to be addressed.

5 Summary and Recommendations

Here, we summarize the major gains and (at least initial) losses that we would experience by switching from the current autotools-based build system for Trilinos to the prototype CMake-based build system for Trilinos. Then, separately, we summarize the gains and losses that would be experienced in switching from the current Perl-based test harness for Trilinos to the prototype CMake/CTest/CDash-based test harness for Trilinos. We list these separately because we can decide to replace the current autotools build system with the CMake system and still maintain the current Perl-based test harness.

5.1 Gains and losses for switching from autotools to CMake for the build system

1. What we gain:
   (a) Full dependency tracking of every kind possible on all platforms (i.e. header to object, object to library, library to executable, and build system files to all built files).
   (b) Support for shared libraries on a variety of platforms.
   (c) Support for MS Windows (i.e. Visual Studio projects, Windows installers, etc.).
   (d) Simplified build system and easier maintenance (extremely easy to add new packages and maintain existing packages).
   (e) Improved mechanism for extending capabilities (as compared to M4 in autotools).
   (f) Ability to affect the development of the build tools with good existing collaborations (i.e. with both Kitware and with organization 1420).
   (g) Significant “in house” knowledge-base (i.e. visualization group in 1420).
   (h) One hundred percent automated intra-package dependency tracking and handling (built into the prototype Trilinos/CMake build system).
2. What we lose (at least initially):
   (a) CMake requires that all uses have ’cmake’ installed on their machine when building from source and users will need to have at a very recent version of cmake. (However, cmake is very easy to build from source).
   (b) Support for circular test/example and package libraries is not provided in the current prototype Trilinos/CMake build system.

5.2 Gains and losses for switching the test harness to the current system to CTest/CDash

1. What we gain:
   (a) Test time-outs (this is a major maintenance issue for the current Perl-based test harness).
   (b) Memory testing with Valgrind and purify that is backed up by Kitware and a larger development community.
   (c) Line coverage testing that is backed up by Kitware and a large development community.
   (d) Support for selecting and excluding subsets of tests based on regular expressions (but better support for keywords would be welcomed).

(e) Better integration with the build system (e.g. easier to support more advanced features like PBS batch systems and flexible testing control).

(f) Better tracking of specific tests (i.e. each and every test can have a unique name that is easy to find).

2. What we lose (at least initially):

   (a) Separate reporting of test results for different Trilinos packages on the web page and in emails sent out (however, such support could be layered on top of CTest and CDash).

   (b) Support for selectively disabling package tests/examples and entire packages when a build fails (however, such support could be layered on top of CTest for driving the test harness).

5.3 Final recommendations

The potential gains for switching from the current autotools-based build system for Trilinos to the prototype CMake-based build system summarized above are overwhelming. Therefore, our recommendation is to transition all of Trilinos to the new CMake-based build system and completely drop the current autotools build system as soon as possible. However, maintaining limited support for the current autotools-based build system through the next major release of Trilinos would be recommended.

There are also significant advantages to supporting the CMake/CTest/CDash-based test harness as summarized above. However, some features supported by the home-grown Perl-based test harness will take considerable time and effort to replicate with the CMake/CTest/CDash system. Therefore, our recommendation is to maintain test suites for both the new CMake/CTest/CDash system and the current Perl-based test harness until such time that the infrastructure around the CMake/CTest/CDash system sufficiently supports the compartmentalization of test results for archiving and reporting. This means maintaining each package's test/definition file, and adding TRILINOS_ADD_TEST(...) calls in CMakeLists.txt files. By maintaining both testing systems, we will have the best of both worlds but at the cost of needing to maintain two test systems for some time. However, it is expected that maintaining both testing system will be much easier that maintaining both build systems for several reasons.

The current Trilinos/CMake prototype build system is now at a state where we believe it can now be pushed out to all of Trilinos very rapidly.

References

[1] Ken Martin and Bill Hoffman. *Mastering CMake: A Cross-Platform Build System*. Kitware Inc, fourth edition, 2007.

DISTRIBUTION:

1 MS 0899 Technical Library, 9536 (electronic)


Sandia National Laboratories